Theoretical Computer Science 412 (2011) 1243–1262

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Highly concurrent multi-word synchronization☆,☆☆

Hagit Attiya*, Eshcar Hillel

Department of Computer Science, Technion, Israel

ARTICLE INFO

Article history:
Received 22 July 2008
Received in revised form 12 December 2010
Accepted 19 December 2010
Communicated by G. Ausiello

Keywords:
Multi-word synchronization
Concurrent data structures
Local nonblocking
DCAS

ABSTRACT

The design of concurrent data structures is greatly facilitated by the availability of synchronization operations that atomically modify k arbitrary items, such as k-*read–modify–write* (kRMW). Aiming to increase concurrency in order to exploit the parallelism offered by today's multi-core and multi-processing architectures, we propose a highly concurrent software implementation of kRMW, with only constant space overhead. Our algorithm ensures that two operations delay each other only if they are within distance $O(k)$ in the *conflict graph*, induced by the operations' data items.

The algorithm uses *double compare-and-swap* (DCAS). When DCAS is not supported by the architecture, the algorithm of Attiya and Dagan (2001) [3] can be used to replace DCAS with (unary) CAS, with only a slight increase in the interference among operations.

1. Introduction

Multi-word synchronization operations, like k-*read–modify–write* (kRMW), allow one to read the contents of several data items, compute new values and write them back, all in one atomic operation. A popular special case is k-*compare-and-swap* (kCAS), where the values read from the data items are compared against specified values, and if they all match, the items are updated. Multi-word synchronization facilitates the design and implementation of concurrent data structures, making it more effective and easier than when using only single-word synchronization operations. For example, removing an item from a doubly linked list can easily be implemented if 3CAS is used to acquire the item and its neighbors; a right or left rotation applied on a node in an AVL tree can easily be implemented if 4CAS is used to acquire the node, its parent and its two children.

Today's multi-core architectures, however, support in hardware only single-word synchronization operations like CAS or LL/SC, or at best, *double compare-and-swap* (DCAS). Thus, kRMW or kCAS must be provided in software.

It is crucial to allow many operations to make progress concurrently and complete without interference, in order to utilize the capabilities of contemporary architectures. Clearly, when operations need to simultaneously access the same items, an inherent "hot spot" is created and operations cannot proceed concurrently. However, typical implementations of kRMW create an additional, not obviously necessary, delay, when the progress of an operation is hindered due to operations that do not conflict on the same items. In these implementations, e.g., [25,19,13,5,26], an operation tries to acquire all the items it needs, one by one; if another operation already acquired an item, the operation is blocked and can either *wait* for the item to be released (possibly while *helping* the conflicting operation to make progress) or *reset* the conflicting operation and try to acquire the item. In these schemes, *chains* of operations delaying each other may be created (see Section 2). It is

☆ The preliminary version of this paper has appeared in the proceedings of ICDCN 2008. The paper has been invited for publication in TCS as one of the best papers presented at the conference after being subjected to the standard TCS refereeing procedure.

☆☆ Supported by a grant from Intel Corporation and the *Israel Science Foundation* (grants 953/06 and 1227/10).

* Corresponding author.
E-mail address: hagit@cs.technion.ac.il (H. Attiya).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.12.049

possible to construct recurring scenarios where an operation is delayed a number of steps proportional to the length of such a chain, causing a lot of work to be invested, while only a few operations complete.

These considerations can be described more precisely through the *conflict graph* of operations that overlap in time; in this graph, vertices represent operations, and an edge connects two vertices if they access the same data item. The *distance* between two operations in a conflict graph is the length of the shortest path between them. For example, simultaneous operations accessing the same data item are at distance one in the conflict graph.

In *disjoint-access parallel* implementations [19], operations that are not connected in the conflict graph do not delay each other, namely, they wait for or help only (transitively) conflicting operations. However, even when operations choose their items uniformly at random, it has been shown [11], both analytically and experimentally, that paths in the conflict graph have non-constant length, depending on the total number of operations. This means that even if the implementation is disjoint-access parallel, an operation can be delayed by "distant" operations.

These adverse effects can be mitigated if operations are delayed only due to operations within fixed distance. Informally, an implementation is *d-local nonblocking* if whenever an operation *op* takes an infinite number of steps, some operation within distance d from *op* completes. This implies that the throughput of the algorithm is localized in components of diameter d in the conflict graph, and operations are effectively isolated from operations at distance $> d$.

Being $O(k)$-local nonblocking guarantees that in every execution, some operation completes after a finite number of steps of some process.

Our contribution. We present an $O(k)$-local nonblocking implementation of kRMW that stores a constant amount of information (independent of k), in each data item.

Our main new algorithmic ideas are explained in the context of a *blocking* implementation, BLocalRMW (Section 4), in which the delay of an operation may block only operations that access *nearby* data items; operations that access data items that are farther than $O(k)$ away in the conflict graph are not affected. This is a variant of the *failure locality* property [6] (see Definition 1).

A key algorithmic idea is that the effect of delays can be bounded, yielding better concurrency, if an operation decides whether to wait for another operation or reset it by comparing how advanced they are in obtaining their data items. If the conflicting operation is more advanced, the operation waits; otherwise, the operation resets the conflicting operation and seizes the item.

While a similar approach has been used in many resource allocation algorithms, dating back to the classical *wound-die* and *wound-wait* deadlock prevention schemes [21], it is not at all obvious that it ensures locality. In particular, operations may repeatedly reset each other, without any operation completing within $O(k)$ distance. One contribution of our paper is in analytically bounding the locality properties of this approach; a challenging part of the proof shows that an operation cannot be repeatedly reset, without some operation completing in its $O(k)$-neighborhood. Our proof uses a *potential* method to show progress in the neighborhood of any operation *op*, even if it is repeatedly reset. That is, we maintain a *potential vector* in which each entry indicates the number of items acquired by an operation in the $O(k)$-neighborhood of *op*; we show that the vector increases with each reset of *op*, and therefore, eventually some operation, in the $O(k)$-neighborhood of *op*, acquires all its data items and completes.

Another important contribution is in handling the symmetric situation, when overlapping operations that have made the same progress, i.e., acquired the same number of items, create a chain in the conflict graph. Symmetry can be broken, in principle, by relying on operation identifiers, so as to avoid deadlocks and guarantee progress. Doing so, however, may create a long chain (which may involve all processes). To avoid these delays, we break ties by having the conflicting operations try to atomically acquire the two objects associated with the operations, using *double compare-and-swap* (DCAS). This efficiently partitions symmetric chains into disjoint constant-length chains, ensuring that operations are delayed only due to close-by conflicts. This is the only scenario in which DCAS is employed, and the less common these scenarios are, the less frequently DCAS is used.

We finally present LocalRMW (Section 5), which guarantees progress even when processes stop taking steps by *helping* a blocking operation that is more advanced, instead of waiting for it to complete; we still reset conflicting operations that are less advanced. The proof of this algorithm shows how helping mitigates the impact of process failures and proves, in a manner similar to the proof of BLocalRMW, that LocalRMW is $O(k)$-local nonblocking.

Using a $O(\log^* n)$-local nonblocking implementation of DCAS from CAS [3], yields a CAS-based kRMW implementation that is $O(k + \log^* n)$-local nonblocking.

2. Related work

The first implementation of multi-word synchronization that use helping are the "locking without blocking" schemes [5, 26], where operations recursively help other operations, without releasing the items they have acquired; these algorithms are $O(n)$-local nonblocking. The static *software transactional memory* (STM) [25] also implements multi-word synchronization. Operations acquire items by the order of their memory addresses, and help only operations at distance 1. Nevertheless, it is $O(n)$-local nonblocking, as demonstrated by the following example. Consider a chain of overlapping two-item operations, in which all operations acquiring their low-address data items successfully, then failing to acquire their high-address data items, which are held by the next operation in the chain. Then all operations help their (immediate)

Table 1
Comparison of multi-word synchronization algorithms, showing locality and progress properties, as well as the space complexity per item. The table also indicates whether the algorithm uses DCAS, and whether the algorithm is or can be made to be dynamic.

Algorithm	Locality	Progress	Space per item	Uses DCAS	Dynamic
Turek et al. [26]	$O(n)$	Local nonblocking	$O(1)$	No	No
Barnes [5]	$O(n)$	Local nonblocking	$O(1)$	No	No
Shavit and Touitou [25]	$O(n)$	Blocking	$O(1)$	No	No
Afek et al. [1]	$O(k + \log^* n)$	Local nonblocking	$O(k)$	No	No
Harris et al. [13]	$O(n)$	Local nonblocking	$O(1)$	No	Yes
Herlihy et al. [14]	$O(n)$	Obstruction free	$O(1)$	No	Yes
BLocalRMW (this paper)	$O(k)$	Blocking	$O(1)$	Yes	Can be
BLocalRMW using [3]	$O(k + \log^* n)$	Blocking	$O(1)$	No	Can be
LocalRMW (this paper)	$O(k)$	Local nonblocking	$O(1)$	Yes	Can be
LocalRMW using [3]	$O(k + \log^* n)$	Local nonblocking	$O(1)$	No	Can be

neighbor. Prior to helping, the operations relinquish their items, thus the operations discover that their help is unnecessary. After the last operation in the chain completes, again all other operations may acquire their low-address data items, and try, in vain, to help the next operation in the chain, which releases its items due to its neighbor, etc. As the length of the chain of overlapping operations increases, the number of times an operation futilely helps another operation increases as well.

Afek et al. [1] present a CAS-based implementation of kRMW that is $O(k + \log^* n)$-local nonblocking,[1] like the CAS-based version of LocalRMW. Their implementation works recursively in k, going through the items according to their memory addresses, and coloring the items before proceeding to acquire them; at the base of the recursion (for $k = 2$), it employs the DCAS implementation of Attiya and Dagan [3]. Due to its recursive structure, the algorithm is quite complicated, making it hard to derive detailed pseudocode and correctness proof, which are not provided in their paper.

The recursive structure of the implementation of [1] requires to store $O(k)$ information in each data item, and to hard-wire k, uniformly for all operations. In contrast, LocalRMW stores a fixed amount of information per data item, regardless of k; in fact, it can be modified so that each operation accesses a different number of data items. Moreover, the implementation of [1] acquires items in increasing order and performs preparatory calculation (coloring) on them, implying that it must receive all items when it starts; thus it is inherently *static*, and must receive all data items in advance. In contrast, our implementation does not depend on the memory addresses of the items and can be modified to work when data items are given one-by-one. (We do not present these extensions as they obfuscate our key ideas.)

Other implementations of dynamic multi-word synchronization operations, such as [13], use recursive helping, and are $O(n)$-local nonblocking. DSTM [14] provides an obstruction-free multi-word synchronization, which is dynamic and does not use helping: a blocked transaction releases its items and retries. However, DSTM has $O(n)$ failure locality in the scenario given for [25], modified so that instead of completing, the transaction at the end of the chain stops taking steps. In this scenario, transactions that stop taking steps can cause a transaction at distance $O(n)$ to retry over and over again.

Table 1 summarizes this comparison.

Several STMs use a designated *contention manager* for deciding how to handle conflicts. Like our BLocalRMW and LocalRMW, some contention managers, e.g., SizeMatters [22], Karma and Polka [23], arbitrate between conflicting transactions based on the number of acquired items or bytes accessed. These contention managers, however, do not address symmetry breaking in the case of equal progress, and scenarios similar to the one given for [25] can create long *delay chains*. Also, they neither state nor prove analytical bounds on their progress and locality.

Schneider and Wattenhofer [24] evaluate a contention manager by its *makespan*, i.e., the total execution time of all operations. Their analysis implies that in our example of overlapping operations, a chain of length n can yield $O(n)$ makespan. They use randomization to break symmetry and improve the locality of contention management, but their algorithm is still with high probability $O(\log n)$ away from the optimum, thus not having constant locality.

3. Preliminaries

We consider a standard model for a shared memory system [4] in which a finite set of *asynchronous processes* $p_1, \ldots, p_n$ communicate by applying *primitive* operations to shared *memory locations* $l_1, \ldots, l_m$. A *configuration* specifies the local state of each process and the value of each memory location. In the (unique) *initial configuration*, every process is in its initial state and every location contains its initial value.

An *event* is a step in which a process executes some local computation and applies a primitive to the memory. In addition to standard READ and WRITE primitives, we employ CAS(l_j, *exp*, *new*), which writes the value *new* to location l_j if its value is equal to *exp*, and returns a success or failure indication, and a DCAS primitive, which is similar to CAS, but is applied to two memory locations atomically (see Fig. 1).

[1] They state $O(\log^* n)$-local complexities, treating k as a constant.

```
boolean CAS(l, exp, new) {
    // Atomically
    if l = exp then
        l ← new

        return TRUE
    return FALSE
}
```

```
boolean DCAS(l[2], e[2], n[2]) {
    // Atomically
    if l[1] = e[1] and l[2] = e[2] then
        l[1] ← n[1]
        l[2] ← n[2]
        return TRUE
    return FALSE
}
```

Fig. 1. The *compare&swap* (left) and *double compare&swap* (right) primitives.

$$
\begin{aligned}
op_1 &= 3\text{RMW}(v_1, v_2, v_6)\\
op_2 &= 2\text{RMW}(v_2, v_5)\\
op_3 &= 2\text{RMW}(v_1, v_3)\\
op_4 &= 2\text{RMW}(v_3, v_4)\\
op_5 &= 2\text{RMW}(v_1, v_4)
\end{aligned}
$$

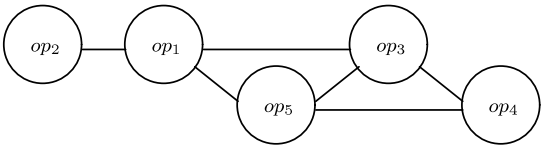


Fig. 2. The conflict graph of five overlapping operations; data items are shown on left.

An *execution interval* $\alpha = C_0, \phi_0, C_1, \phi_1, C_2, \ldots$ is a finite or infinite alternating sequence, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \ldots$. An *execution* is an execution interval in which C_0 is the initial configuration.

An *implementation* of a kRMW operation specifies the data representation of the operations and the data items, and provides an algorithm, defined in terms of primitives, that processes follow in order to execute the operation.

The *interval of an operation op* is the execution interval between the first event and last event of the algorithm executed by the process invoking *op*. If the execution does not include a last event then the interval of *op* is the suffix of the execution, starting in the first event of *op*; the interval can be infinite. Two operations *overlap* if their intervals overlap.

We require the implementation to be *linearizable* [16], that is, any execution can be extended by discarding some pending operations and completing the others, such that the extended execution is equivalent to some serial execution, called its *linearization*, which preserves the order of non-overlapping operations.

The *conflict graph* of a configuration C is an undirected graph, in which vertices represent operations, and edges connect two operations if their data sets intersect. The graph captures the distance between overlapping operations: If C is a configuration during the intervals of operations op_i and op_j, accessing the same data item, the graph includes an edge connecting the vertices op_i and op_j. Fig. 2 gives an example for the conflict graph of a configuration.

The conflict graph of an operation *op* is the union of the conflict graphs of all configurations C during *op*'s interval; that is, the vertices are the union of the vertices of all these conflict graphs, and similarly for edges.

The *distance* between two operations, *op* and *op*′, is the length (in edges) of the shortest path between *op* and *op*′ in the conflict graph. If the operations access a common item, then their distance is one; if there is no path between the operations, then their distance is ∞. The *d-neighborhood of an operation op* contains all the operations at distance $\leq d$ from *op* in the conflict graph of its interval.

Definition 1. An algorithm has $\langle d_1, d_2\rangle$*-failure locality* if in every execution, some operation in the d_1-neighborhood of *op* invoked by process p completes after a finite number of steps of p, unless a process that invoked an operation in the d_2-neighborhood of *op* stops taking steps.[2]

We often abbreviate and say that an algorithm has d *failure locality*, with $d = \max(d_1, d_2)$.

The next definition is an analogue of failure locality guaranteeing progress in a neighborhood of a given diameter, even when processes stop taking steps.

Definition 2. An algorithm is *d-local nonblocking* if in every execution, some operation in the d-neighborhood of *op* invoked by process p completes after a finite number of steps of p.

4. BLocalRMW: a blocking algorithm with 3*k* failure locality

BLocalRMW follows a simple high-level scheme in which an operation tries to acquire all its items, one by one, and applying its changes to these items only while holding all of them.

An important aspect of the algorithm is in handling situations in which an operation op_1 finds that an item it needs is already acquired by another, *blocking* operation op_2. BLocalRMW uses the number of data items that are already acquired to decide whether op_1 *waits* for or *resets* op_2, releasing all the items acquired by op_2, and seizing the required item. The operation op_1 waits for op_2 only if op_2 is more advanced in acquiring its items, i.e., op_2 has acquired more items. Otherwise, if op_2 has acquired fewer items than op_1, then op_1 resets op_2.

2 The original definition of Choy and Singh [6] requires an operation to complete if no operation fails in its d-neighborhood, while we only guarantee that *some* operation in the neighborhood completes. This is analogous to the distinction between *starvation-free* and *deadlock-free* algorithms.

Pseudocode 1 *k*-Read-Modify-Write: High-level algorithm.

```
rmw(Item items[k]) {
    WRITE(dataset, items)
    while READ(modifyDone) = FALSE do
        cnfl ← READ(conflict)
        ctx ← READ(context)
        executeIteraion(ctx, cnfl)
}

executeIteraion(Context ctx, Conflict cnfl) {
    if cnfl.blocking = ⊥ then
        tryAdvancingOp(ctx, cnfl)
    else
        handleConflict(ctx, cnfl)
}

tryAdvancingOp(Context ctx, Conflict cnfl) {
    if ctx.counter < k then
        acquireNextItem(ctx, cnfl)
    else
        applyChangesAndRelease(ctx)
}

acquireNextItem(Context ctx, Conflict cnfl) {
    if acqOperation(self, ctx, ctx.round) then
        item ← READ(dataset[ctx.counter])
        if acqItem(item, ctx, cnfl) then
            increaseCounter(ctx)
        else
            initializeConflictInfo(ctx, cnfl)
}
```

```
structure Owner {Operation op, int round}
structure Context {int counter, Owner owner, int round}
structure Conflict {int counter, int round, Owner blocking}

class Item {
    Data data
    Owner owner
}

class Operation {
    Item dataset[k]
    Context context
    Conflict conflict
    boolean modifyDone
}
```

Fig. 3. Data structures and classes for BLocalRMW.

An operation resets another operation only after acquiring ownership over it. Resetting an operation only involves releasing the items it acquired, and does not involve rollback, as the operation has not applied any changes yet.

Another important aspect of BLocalRMW is in handling the *symmetric* case, when op_1 and op_2 have acquired the same number of items, by applying DCAS to atomically acquire ownership of both operations; the operation that acquires ownership, resets the other operation. This breaks apart long hold-and-wait *chains* that would deteriorate the locality as well as hold-and-wait *cycles* that can cause a deadlock.

The high-level scheme appears in Pseudocode 1.

The execution of an operation is partitioned into *rounds*, each starting when the operation is reset. At each round, the operation tries to acquire its data items; if the operation is reset all the items acquired by the operation are released; in the last round, the operation succeeds in acquiring all the items, applies its changes, and finally, releases all its items.

Rounds are divided into iterations (executeIteraion); at each iteration the operation either tries to make progress (tryAdvancingOp), by either acquiring an additional item (acquireNextItem) or applying the changes to the data items it has acquired (applyChangesAndRelease), or it handles a conflict (handleConflict method). When acquiring an item, the operation increases its counter, otherwise, it discovers that another operation owns the item, and initializes the conflict information attribute, so the conflict can be handled in the next iteration.

4.1. Data structures

BLocalRMW is derived from the general scheme by substituting the methods for the high-level scheme of Pseudocode 1. Its data structures appear in Fig. 3. Memory locations are grouped in contiguous blocks, called *item objects*. Each item object contains a *data* attribute and an *owner* attribute, including the operation, *op*, and the *round* in which the operation acquired the item; the owner attribute is $\perp$ if no operation has acquired the data item.

For each operation, we maintain an *operation object*[3] containing a *dataset*, referencing the set of items the operation has to access and modify; it is initialized when the operation is first invoked. The *context* attribute is a tuple of a *counter* holding the number of items acquired so far (initially 0), an *owner* and the *round number* of the operation (initially 0). An operation object also contains some local attributes, which are not shared and are only visible to the process executing the operation.[4] The *modifyDone* flag indicates whether or not the modifications of the operation have been applied. The *conflict* attribute stores information about a conflict, if there is any: the owner of the item *blocking* the operation, and the *counter* and *round* values of the blocked operation.

4.2. Implementation

The methods of BLocalRMW appear in Pseudocode 2 that is described next.

3 These are similar to *transaction descriptors* used in some STM implementations, e.g., [12,14].

4 This will be changed later, when we present LocalRMW.

Pseudocode 2 Methods for BLocalRMW

```
 1: applyChangesAndRelease(Context ctx) {
 2:     modify()
 3:     modifyDone ← TRUE
 4:     releaseDataset()
 5: }

 6: boolean acqOperation(Operation op, Context ctx, int rnd) {
 7:     CAS(op.context, ⟨ctx.counter,⊥,ctx.round⟩, ⟨ctx.counter,⟨self,rnd⟩,ctx.round⟩)
 8:     return op.ctx.owner ≠ ⊥ and op.ctx.owner.op = self
 9: }

10: boolean acqItem(Item item, Context ctx, Conflict cnfl) {
11:     CAS(item.owner, ⊥, ⟨self,ctx.round⟩)                          // acquire the current item
12:     owner ← READ(item.owner)
13:     return owner ≠ ⊥ and owner.op = self                          // the item is owned by this operation
14: }

15: increaseCounter(Context ctx) {
16:     WRITE(context, ⟨ctx.counter+1,ctx.owner,ctx.round⟩)
17: }

18: initializeConflictInfo(Context ctx, Conflict cnfl) {
19:     conflict ← ⟨ctx.counter,ctx.round,owner⟩
20: }

21: handleConflict(Context ctx, Conflict cnfl) {
22:     blkOp ← conflict.blocking.op
23:     blkCtx ← READ(blkOp.context)
24:     if ⟨ctx.round,blkCtx.round⟩ ≠ ⟨round,conflict.blocking.round⟩ then
25:         conflict ← ⟨⊥, ⊥, ⊥⟩                                      // reset conflict
26:         return
        // conflict with an operation with a lower counter
27:     if conflict.counter > blkCtx.counter then
28:         if CAS( blkOp.context, ⟨blkCtx.counter,⊥,blkCtx.round⟩, ⟨blkCtx.counter,⟨self,ctx.round⟩,blkCtx.round⟩)
            then reset(blkOp)
29:         return
30:     WRITE(context, ⟨ctx.counter,⊥,ctx.round⟩)                     // release this operation object
        // conflict with an operation with an equal counter
31:     if conflict.counter = blkCtx.counter then
32:         if DCAS( context,                    blkOp.context,
                     ⟨ctx.counter,⊥,ctx.round⟩,  ⟨blkCtx.counter,⊥,blkCtx.round⟩,
                     ⟨ctx.counter,⟨self,ctx.round⟩,ctx.round⟩,  ⟨blkCtx.counter,⟨self,ctx.round⟩,blkCtx.round⟩)
            then reset(blkOp)
        // conflict with an operation with a higher counter
33:     if conflict.counter < blkCtx.counter then No-op
34: }

35: reset(Operation blkOp) {                                          // resetting another blocking operation
36:     ctx ← READ(context)
37:     item ← READ(dataset[ctx.counter])                             // the item held by the blocking operation
38:     WRITE(item.owner, ⟨self,ctx.round⟩)                           // seize the item
39:     WRITE(context, ctx.counter+1,ctx.owner,ctx.round)             // increase counter
40:     blkOp.releaseDataset()                                        // release the data set of the blocking operation
41: }

42: releaseDataset() {
43:     ctx ← READ(context)
44:     for i = 0 . . .ctx.counter−1 do
45:         item ← READ(dataset[i])                                   // an item to release
46:         WRITE(item.owner, ⊥)                                      // release the item
        // release operation object, reset counter, increase round number
47:     WRITE(context, ⟨0, ⊥,ctx.round+1⟩)
48: }
```

The applyChangesAndRelease method applies the changes of the operation (line 2), and releases the data items (line 4).

The acqOperation method acquires ownership on the operation by setting the *owner* attribute to the reserved word *self*, denoting the object of the operation whose code is being executed (line 7). To simplify the pseudocode, as well as the correctness proofs, when acquiring an operation object, CAS is applied on the entire *context* and not just on the *owner*.

The handleConflict method reads the identity of the blocking operation op' (line 22), and its context (line 23). Then it checks whether the round numbers of op or op' changed (line 24), indicating that either one or both of them released the items in its data set and the conflict is resolved.[5] If one of the round numbers changes, the method resets the conflict information (line 25) and proceeds to the next iteration (line 26). Otherwise, it compares the counter of op with the counter of op'. If the counter of op is higher than (line 27) or equal to (line 31) the counter of op', the method tries to reset op' (lines 28, 32) so as to seize the item. For this purpose, op needs to hold the operation objects of both op and op'. When the counter of op is higher than the counter of op', the method keeps the operation object of op, and tries to acquire the operation object of op' (line 28), using CAS suffices in this case. When the counters are equal, the method releases the operation object of op (line 30) and tries to atomically acquire both operation objects (line 32) by applying DCAS.

The handleConflict method releases the operation object of op (line 30) also if the counter of op is lower than the counter of op' (line 33), but does nothing before the next iteration (line 33). Note that after releasing the operation object, op reacquires the operation object (and resumes the execution of the operation) only when it is no longer blocked by op' (since one of the round numbers changed) or when it is about to reset op'. This allows other operation to reset op if op is blocking them (see the proof of Lemma 5).

The reset method reads the item held by the blocking operation (line 37), and then seizes it by overriding the value written in the item's owner field (line 38). Then, it increases the counter of the (blocked) operation (line 39) and releases the data set of the blocking operation (line 40).

The releaseDataset method reads (line 45) and releases (line 46) every item that was acquired by the operation, and resets the context of the operation by setting its counter to zero and its owner to $\perp$, and increasing the round number (line 47).

4.3. An execution example

Fig. 4 shows an example of a chain of conflicts between the operations from Fig. 2. In Fig. 4(a) op_2 holds v_5 and needs to acquire v_2; op_1 holds v_2 and v_6 and it needs to acquire v_1; op_3, op_4, and op_5, hold v_1, v_3, and v_4 respectively and need to acquire the successor item in the loop. The counter of op_1 is 2, and the counters of op_2, op_3, op_4, op_5, are 1. Therefore, the operation op_1 is blocked by an operation op_3 (with a lower counter) and tries to reset op_3. In Fig. 4(b) op_1 completes the reset, and acquired all its data items (the counter is 3), and can apply its changes and release the items. Operation op_2 holds v_5 and is blocked by op_1 on v_2 (with a higher counter). Hence, op_2 releases its own operation object, and waits for op_1 to release v_2.

The conflicts between op_3, op_4 and op_5 may lead to a cycle of hold-and-wait operations, since they have equal counters. Fig. 4(c) shows how DCAS breaks the symmetry and avoids a deadlock: the processes release their operation objects; op_3 tries to atomically acquire the operation objects of op_3 and op_4, op_4 tries to acquire the operation objects of op_4 and op_5, and op_5 tries to acquire the operation objects of op_5 and op_3. In Fig. 4(c), only op_5 succeeds to acquire ownership on its operation object and the operation object of op_3, and it resets op_3.

4.4. Proof of correctness

Note that a process resets an operation only if it owns the operation's object. Since a process owns its own operation object when it has acquired all its items, it is not reset anymore, and its changes are applied in isolation. This can be used to prove that BLocalRMW is linearizable; the full proof is omitted since it is a simplified version of the safety proof for LocalRMW (see Section 5.3). We concentrate on proving the locality of the algorithm by bounding the length of delay chains.

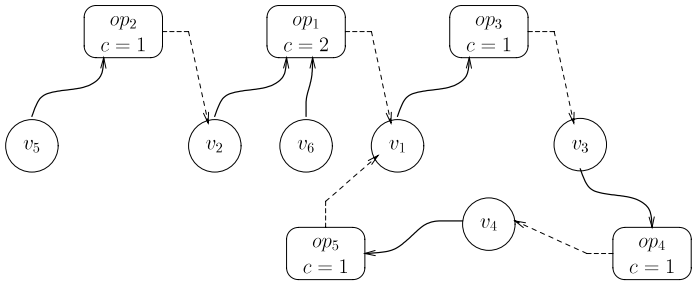
One type of delay chain is created when a process stops taking steps and causes operations of other processes to be blocked; we show, in Lemmas 5 and 6, that the length of such chains is $O(k)$.

More intricate, and less intuitive, delay chains are created when operations reset other operations. For example, assume an operation op_1 is reset by another operation op_2, then, a third operation resets op_2. At some later time, the processes executing op_2 and op_1 can reacquire their operation object, and the same scenario may happen over and over again. It may even seem as if a livelock can happen due to a cycle of resetting operations.

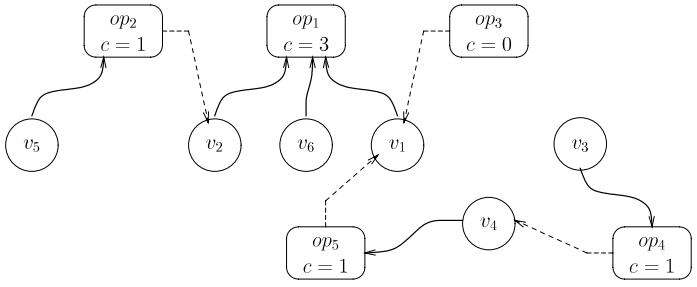
Fig. 5 illustrates how an operation can be reset many times before some operation completes in its k-neighborhood. For $i \geq 1$, after op_i acquires an additional item and increases its counter to $i+1$, it is blocked by op'_{i+1} with equal counter. Initially, op_0 releases its operation object in order to reset op'_1 and seize the item op_0 needs. Instead, op_1 resets op_0, seizes the item op_0 owns, and increases its own counter to 2. Then, op_1 is blocked by op'_2 with equal counter, 2. In a similar way, op_2 resets op_1 after it releases its operation object, and op_3 resets op_2, so that their items are released. Later, op_0 and op_1 are able to reacquire their first items. This recurring reset scenario is repeated with longer chains of resets each time.

However, inspecting the example reveals that the longer the chain is, the higher is the counter of its last operation, and after $k/2$ resets of op_0, op_{k-1} at distance $k-1$ from op_0 owns all its data items, and it can complete. The following lemmas formalize this intuition. Since an item is seized during a reset, an operation makes progress whenever it resets another operation, as stated in the next lemma. Let $c(op)$ be the counter of op in a given configuration.

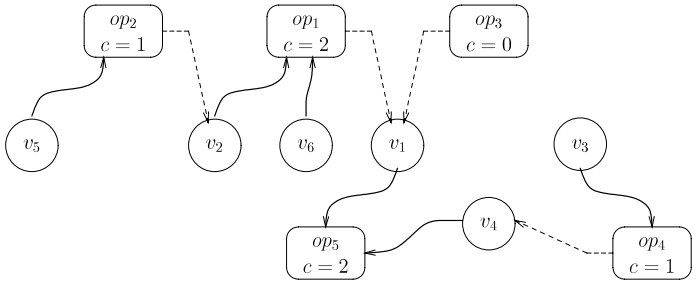
[5] This is similar to the way Shavit and Touitou [25] use the version number to validate the operation.



(a) op_1 and op_5 are blocked by op_3.



(b) A possible scenario after 4(a): op_1 resets op_3.



(c) Another possible scenario after 4(a): op_5 resets op_3.

Fig. 4. Conflicting operations examples. An operation object points to the next item to be acquired (dashed line); c is the value of its counter. Items are owned by the operation to which they point.

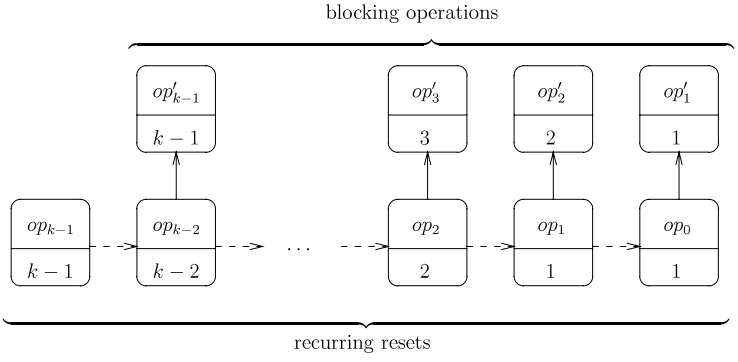


Fig. 5. A recurring resets scenario. The number below an operation indicates its counter; solid arrows indicate blocking and dashed arrows indicate reset. Initially, for $i \geq 1$, op_i has counter i; for $i \geq 0$, op_i owns the item op_{i+1} needs next; and op_0 is blocked by op'_1 with equal counter, 1. For $i \geq 1$, after op_i acquires the next item and increases its counter, it is blocked by op'_{i+1} with equal counter $i+1$.

Lemma 1. *If an operation op_1 resets another operation op_2, then $c(op_1) = m_1 \geq c(op_2) = m_2$ in the configuration in which the reset is invoked, and $c(op_1) = m_1 + 1 > m_2$ in the configuration after the reset completes.*

Proof. Before invoking reset, op_1 compares the counters of op_1 and op_2 (lines 27 or 31). If $c(op_2) < c(op_1)$ (line 27), then op_1 tries to acquire the operation object of op_2 (line 28). The counter of op_1 does not change while op_1 is holding its operation object. If op_1 acquires the operation object of op_2, then the counter of op_2 also has not changed since op_1 read it.

If the counters are equal (line 31), then op_1 acquires the operation objects of op_1 and op_2 (line 32), before resetting op_2 (line 32). Similar arguments imply that the counters do not change.

In reset, op_1 increases its counter (line 39), and thus, $c(op_1) = m_1 + 1 > m_2$ after it. □

Lemmas 2–4 prove that whenever an operation op is reset, some operation in its neighborhood makes progress, e.g., increasing the counter or completing, and that after op is reset a bounded number of times, some operation in its neighborhood completes. For this purpose, we define a dynamic set of operations, $\mathcal{R}_{op}$. Whenever an operation in the $(k-1)$-neighborhood of op completes, $\mathcal{R}_{op}$ is set to the empty set. Whenever an operation $op_j \in \mathcal{R}_{op} \cup \{op\}$ is reset by an operation op_i, op_j is removed from $\mathcal{R}_{op}$ (if it is in $\mathcal{R}_{op}$), and op_i is added to $\mathcal{R}_{op}$ (if it is not already in $\mathcal{R}_{op}$).

Lemma 2. *Every operation op_j in $\mathcal{R}_{op}$ is at distance $\leq c(op_j) - 1$ from op.*

Proof. The proof is by induction on the length of the execution interval of op. The base case is when the operation starts; in this case, $\mathcal{R}_{op}$ is empty and the claim vacuously holds.

For the induction step, assume that the claim holds for some prefix of op's execution interval, and consider first a step that does not change $\mathcal{R}_{op}$. If the counter of some operation in $\mathcal{R}_{op}$ increases, then the lemma holds by the inductive assumption. $\mathcal{R}_{op}$ does not change, and by definition, no operation in the $(k-1)$-neighborhood completes. By the inductive assumption, all operations in $\mathcal{R}_{op}$ are in the $(k-1)$-neighborhood of op, thus no operation in $\mathcal{R}_{op}$ completes. Since no operation in $\mathcal{R}_{op}$ is reset, the counters of operations in $\mathcal{R}_{op}$ do not decrease.

Consider now a step that changes $\mathcal{R}_{op}$. If some operation in the $(k-1)$-neighborhood completes and $\mathcal{R}_{op}$ is set to empty, then the claim vacuously holds. Otherwise, an operation op_j resets an operation $op_i \in \mathcal{R}_{op} \cup \{op\}$; $c(op_i) = m \geq 1$ before the reset, since op_i owns at least one item. By the inductive assumption, op_i is at distance $\leq m-1$ from op. Therefore, op_j is at distance $\leq m$ from op, and by Lemma 1, $c(op_j) > m$ after the reset, and the lemma holds. □

The *potential vector of* $\mathcal{R}_{op}$ in a configuration C is a vector with n entries holding the counters of operations in $\mathcal{R}_{op}$. Assume $\mathcal{R}_{op}$ contains r operations, then entries $0 \ldots r-1$ hold the counters of the operations in $\mathcal{R}_{op}$ in decreasing order, and entries $r \ldots n-1$ are all 0's. Note that the entries of the vector are reordered every time the counters are changed, so that all 0's are shuffled to the end of the vector, and other counters appear in decreasing order at the beginning of the vector. We compare vectors in lexicographic order.

Lemma 3. *The potential vector of $\mathcal{R}_{op}$ grows with each reset of op, and it does not decrease unless some operation within op's $(k-1)$-neighborhood completes.*

Proof. Assume that no operation within the $(k-1)$-neighborhood of op completes. By Lemma 2, the operations in $\mathcal{R}_{op}$ are in the $(k-1)$-neighborhood of op, and hence, none of them completes.

If the counter of some operation op' in $\mathcal{R}_{op}$ increases, then the potential vector grows.

Consider an operation, op'', resetting an operation $op' \in \mathcal{R}_{op} \cup \{op\}$, such that $c(op') = m'$. If both op' and op'' are not in $\mathcal{R}_{op}$ then no operation is removed from $\mathcal{R}_{op}$, and op'' is added to $\mathcal{R}_{op}$. As a result, the potential vector of $\mathcal{R}_{op}$ has an additional nonzero entry. Since the vector holds the counters of the operations in $\mathcal{R}_{op}$ in decreasing order, the potential vector grows.

If $op' \notin \mathcal{R}_{op}$ and $op'' \in \mathcal{R}_{op}$ then $\mathcal{R}_{op}$ does not change. Lemma 1 implies that $c(op'')$ increases after the reset, and the potential vector grows.

If $op' \in \mathcal{R}_{op}$ and $op'' \notin \mathcal{R}_{op}$ then op'' replaces op' in $\mathcal{R}_{op}$. Lemma 1 implies that after the reset $c(op'') > m'$, and the potential vector grows.

Finally, if both op' and op'' are in $\mathcal{R}_{op}$, then op' is removed from $\mathcal{R}_{op}$, and its entry in the vector is set to 0, and no operation is added to $\mathcal{R}_{op}$. However, by Lemma 1, before the reset $c(op'') \geq c(op')$, and $c(op'')$ increases after the reset. Thus, some entry to the left of op' in the potential vector increases, and the potential vector grows (since entries are reordered to appear in decreasing order). □

A *progress step* of an operation op is a step in which either op completes or increases its counter. Let n_d be the number of operations in the d-neighborhood of op. By Lemma 2, all the operations in $\mathcal{R}_{op}$ are in the $(k-1)$-neighborhood of op, thus the size of $\mathcal{R}_{op}$ is at most n_{k-1}. The next lemma uses Lemma 3 to show that after a bounded number of progress steps of op, some operation in its $(k-1)$-neighborhood completes.

Lemma 4. *After at most $n_{k-1}k^2$ progress steps of an operation op, some operation completes in its $(k-1)$-neighborhood.*

Proof. If no operation completes in the $(k-1)$-neighborhood of op (including op), then after $n_{k-1}k(k-1)$ progress steps, op increases its counter $n_{k-1}k(k-1)$ times, which means it is reset at least $n_{k-1}k$ times. By Lemma 3, after each reset of op, the potential vector of $\mathcal{R}_{op}$ increases. By the time the vector increases $n_{k-1}k$ times, all the operations in the $(k-1)$-neighborhood of op (at most n_{k-1} operations, including op) have their counters equal to k and hereafter these operations do not release their operation objects. On one hand, the operations in this neighborhood cannot increase their counters, on the other hand, no operation resets an operation in this neighborhood, which implies that op completes in its $n_{k-1}k(k-1)+1 \leq n_{k-1}k^2$ progress step. □

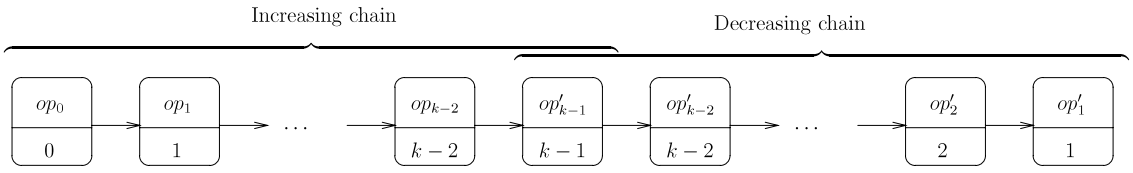


Fig. 6. The operations op_j create an *increasing chain*; each op_j operation is blocked by the operation op_{j+1} holding the data item t_{i+1}. The operations op'_i create a *decreasing chain*; each op'_i operation is blocked by the operation op'_{i-1} holding its operation object.

This shows that recurring resets do not prevent progress in the neighborhood of an operation. We now discuss delay chains created due to processes that stop taking steps.

Consider an operation op that stops taking steps while holding a data item. Another operation op' requiring this item cannot complete without resetting op, which it fails to do if op holds its operation object. If $c(op') > c(op)$, op' keeps its operation object, and it may block a third operation with a higher counter that cannot reset op', and so on.

Formally, an operation op is *m-delayed* by another operation op' when op needs to reset op', $c(op') \leq m$, and some operation other than op has acquired the operation object of op'. In a *decreasing chain* of operations an operation op is m-delayed by the next operation where $m < c(op)$. Since each operation in a decreasing chain has a counter that is strictly lower than the counter of the operation that it blocks, the length of the chain is bounded by k.

The right part of Fig. 6 presents an example of a decreasing chain. The operation op'_2 needs to acquire an item, and is blocked by another operation op'_1 with a lower counter. The operation op'_1 does not complete either because it stops taking steps or because it is repeatedly being reset by other operations, and the operation object of op'_1 is always reacquired before op'_2 has a chance to reset op'_1 and acquire its next item. For every i, $1 \leq i < k-1$, the operation op'_{i+1} is i-delayed by the operation op'_i, since some operation holds the operation object of op'_i while it is blocking op'_{i+1}.

We use the notion of a *big step* [7] to prove that the algorithm has $O(k)$ failure locality. A big step is a measure of time for asynchronous systems. This notion is usually applied to all the processes in the system, and we refine it to describe the progress of a subset of the processes. An execution interval α contains a *big step* of some set of operations S if each operation in S takes at least one step in α. Inductively, an execution interval α contains $s > 1$ *big steps of* S if α can be written as $\alpha'\alpha''$, α' contains $s-1$ big steps of S, and α'' contains a big step of S.

Let l be a constant that is larger than the number of steps a process takes in a single invocation of executeIteration.

Lemma 5. *Consider an operation op that is m-delayed by another operation op′ in a configuration C, $1 \leq m \leq k-1$. Let $\mathcal{DC}$ be a decreasing chain from op′. Then, in an execution interval α starting in C that contains $2l$ big steps of the processes $\{op\} \cup \mathcal{DC}$, one of the following happens:*

1. *Some operation in the $m+2$-neighborhood of op has a progress step, or*
2. *Some operation op″ holds an operation object of another operation in $\mathcal{DC}$.*

Proof. The proof is by induction on m, with a base case at $m = 1$. The operation op is 1-delayed by op', namely when op tries to reset op', $c(op') = 1$ and some operation has acquired the operation object of op'. In this case, $\mathcal{DC} = \{op'\}$.

If op' increases its counter, during α, then Condition 1 holds. If some operation $op'' \neq op'$ holds the operation object of op', during α, then Condition 2 holds. Thus, assume op' holds its own operation object and it is blocked by another operation op'', and $c(op'') \geq c(op')$ (op' cannot be blocked by an operation with a lower counter, since $c(op') = 1$).

The execution interval α contains $2l$ big steps of op and op', during which op' releases its own operation object and some operation reacquires its operation object. If this time, another operation $op'' \neq op'$ acquires the operation object, then Condition 2 holds. Otherwise, op' re-acquires its own operation object. If op' re-acquires its operation object after op'' has completed, then Condition 1 holds. If op' acquires both the operation objects of op' and op'', then op' resets op'' and increases its counter, during α. Otherwise, op' re-acquires the operation object either after op'' is reset and is no longer blocking op', or after op' is reset and is no longer blocked by op''. This implies that during α, after op' re-acquires its own operation object (after releasing it) some operation (possibly, op' itself) in the 1-neighborhood of op' either completes or is reset, increasing the counter of the resetting operation. Thus, some operation in the 2-neighborhood of op (op', or op'') completes or is reset once during α, and some operation in the 3-neighborhood of op has a progress step and Condition 1 holds.

For the induction step, assume the lemma holds for every operation with counter lower than $m > 1$. If an operation other than op' holds the operation object of op', then Condition 2 holds, so assume op' holds its own operation object.

If op' is m'-delayed by an operation op'', $m' < m$, in C, then consider the decreasing chain $\mathcal{DC}'$ from op'' in C. If some operation in the $m+1$-neighborhood of op' (and the $m+2$-neighborhood of op), has a progress step in α, then Condition 1 holds. Otherwise, α contains $2l$ big steps of $\{op'\} \cup \mathcal{DC}' \subset \{op\} \cup \mathcal{DC}$, and by the inductive assumption, some operation owns another operation object in $\mathcal{DC}'$, satisfying Condition 2.

Finally, if op' is m'-delayed by an operation op'', $m' \geq m$, then, as in the base case, some operation in the 3-neighborhood of op has a progress step during α, satisfying Condition 1. □

Another blocking scenario happens when an operation op is blocked by an operation op' whose counter is higher than $c(op)$. Moreover, op' may be blocked by a third operation with a higher counter, and so on, creating an *increasing chain*, also

depicted in Fig. 6 (left side). Since the counter of every operation in an increasing chain is strictly larger than the counter of the operation that it blocks, the length of the chain is at most k.

Decreasing and increasing chains, together with recurring resets may create longer delay chains. In an *increasing–decreasing chain* of operations, every operation op is blocked either by the next operation in the chain, op' with $c(op') > c(op)$, or by a decreasing chain; in the later case, op is m-delayed by the head of the decreasing chain, $m \leq c(op)$.

Lemma 6. *Consider an operation op with $c(op) = m$ in a configuration C, $0 \leq m \leq k$. Let $\mathcal{IDC}$ be an increasing–decreasing chain that includes op. Then, in an execution interval α starting in C that contains $4l$ big steps of $\mathcal{IDC}$, one of the following happens:*

1. *Some operation in $\mathcal{IDC}$ acquires all its data items, or*
2. *Some operation in the $(2k - m)$-neighborhood of op has a progress step, or*
3. *Some operation op'' owns an operation object of an operation $op' \in \mathcal{IDC}$, $op' \neq op''$.*

Proof. The proof is by backward induction on m. In the base case, $m = k$, op acquired all its data set, and Condition 1 holds. For the induction step assume $m < k$, and consider what happens when op executes $2l$ steps, in which it completes at least two iterations of the loop in the rmw method. We inspect the code.

If there is no conflict, then, since the counter of op is not equal to the size of its data set, op tries to acquire its own operation object (line 7). If another operation holds the operation object of op, then Condition 3 holds.

Otherwise, op holds its operation object (line 7 or line 8), and it tries to acquire another item (line 11). Then, op reads the owner of the current item (line 12). If no operation owns the item (line 13), then op may fail to acquire the item since another operation op' owned it but op' no longer owns the item. Hence, op' either completes or is reset, increasing the counter of the resetting operation in the 2-neighborhood of op and Condition 2 holds. If op owns the current data item (line 13), then the counter of op increases (line 16) and Condition 2 holds.

Otherwise, op is blocked by op'; it sets the *conflict* information of the operation object (line 19), and starts a new iteration, in which it handles the conflict. In this iteration, op reads the *context* of op' (line 23). If the round number of op' has changed (line 24), then op' either completes or is reset, increasing the counter of the resetting operation in the 2-neighborhood of op and Condition 2 holds.

If $c(op) > c(op')$ (line 27), then op tries to acquire the operation object of op' and reset it (line 28).

If $c(op) = c(op')$, op releases its own operation object (line 30) and tries to acquire the operation objects of both op and op' and reset op' (line 32).

If $c(op) \geq c(op')$ and op acquires the operation object of op' then Condition 3 holds. If op fails to acquire the operation objects of op', then either some operation other than op holds the operation object of op, satisfying Condition 3, or op is m'-delayed by op', $m' \leq m$, in a configuration C' after $2l$ big steps of $\mathcal{IDC}$. In the latter case, if some operation in the $(2k - m)$-neighborhood of op has a progress step in α, then Condition 2 holds. Otherwise, the suffix of α, α' starting at C' includes $2l$ big steps of the decreasing chain from op' (which is contained in $\mathcal{IDC}$). Hence, Lemma 5 implies that either some operation in the $m + 2$-neighborhood (included in the $(2k - m)$-neighborhood) of op, has a progress step in α', satisfying Condition 2, or some operation owns another operation on the decreasing chain from op', satisfying Condition 3.

If $c(op) < c(op')$ (line 33), the counter of op' is $m' > m$. If some operation in the $(2k - m)$-neighborhood of op has a progress step in α, then Condition 2 holds. Otherwise, α includes $4l$ big steps of the increasing–decreasing chain from op' (which is contained in $\mathcal{IDC}$), and the lemma holds by the inductive assumption. □

After an operation acquires all its items, it holds its operation object until it completes. Hence, other operations cannot reset its counter, and the operation releases its items within $2l$ steps. Thus, if no operation in an increasing–decreasing chain from op (in its $2k$-neighborhood) stops taking steps, then some operation op' in this neighborhood makes enough progress steps and by Lemma 4, some operation in the k-neighborhood of op' and $3k$-neighborhood of op completes. This argument is formalized in the proof of the next lemma.

Theorem 7 (*Failure Locality*)**.** *BLocalRMW has $\langle 3k, 2k \rangle$-failure locality.*

Proof. Assume that the counter of an operation op is m at some configuration C_1 in its execution interval, and consider an increasing–decreasing chain $\mathcal{IDC}_1$ from op in C_1. Consider also the minimum execution interval α_1 starting in C_1 that contains $4l$ big steps of $\mathcal{IDC}_1$ followed by either $2l$ steps of an operation op' on $\mathcal{IDC}_1$, which acquired all its data items, or $2l$ steps of an operation op'' that owns another operation on $\mathcal{IDC}_1$. Since the length of an increasing–decreasing chain is at most $2k - 3$, and no process stops taking steps in the $2k$-neighborhood of op, an execution interval α' that contains $4l$ big steps of $\mathcal{IDC}_1$ exists. If some operation op' in $\mathcal{IDC}_1$ acquired all its data items after α', then op' (which is in the $(2k - 3)$-neighborhood of op) completes within $2l$ steps. Alternatively, if some operation op'' owns another operation on $\mathcal{IDC}_1$ after α', then op'' (which is in the $(2k - 2)$-neighborhood of op) increases its counter within $2l$ steps. Otherwise, by Lemma 6, some operation in the $2k$-neighborhood of op has a progress step, in α_1.

If no operation in the $3k$-neighborhood of op completes during α_1, then we consider an increasing–decreasing chain $\mathcal{IDC}_2$ from op in the configuration after α_1, and an execution interval α_2 from this configuration, which contains a progress step of some operation in the $2k$-neighborhood of op.

In this manner, we define execution intervals $\alpha_3, \alpha_4, \ldots$. Since there are at most n_{2k} operations in the $2k$-neighborhood of op, after at most $n_{2k}n_{k-1}k^2$ such execution intervals, one of these operations op' has $n_{k-1}k^2$ progress steps. Lemma 4

```
structure Owner {Operation op, int round, int aba}
structure Context {int counter, Owner owner, int round}
structure Conflict {int counter, int round, Owner blocking, int aba}
class Item {
    Data data
    Owner owner
}
class Operation {
    Item dataset[k]
    Context context
    Conflict conflict
    boolean modifyDone
}
```

Fig. 7. Definitions of the data and classes structure for LocalRMW.

implies that some operation in the k-neighborhood of op' completes, hence, some operation in the $3k$-neighborhood of op completes. □

5. LocalRMW: A $3k$-Local Nonblocking Algorithm

LocalRMW is a variant of BLocalRMW that uses *helping* to ensure progress in the $3k$-neighborhood of an operation, even when processes stop taking steps. Helping means that when an operation op is blocked by another operation op' with a higher counter, its invoking process p executes op' to ensure it completes and releases the item, instead of waiting for the process p' that invoked op' to do so by itself (which may never happen if p' stops taking steps); we say that p *helps* op' or that op *helps* op'. Helping is *recursive* and if p discovers that op' is blocked by another operation op'', then p also helps op''. Note that op still resets op' if its counter is equal or higher than the counter of op'. In addition, op can be blocked while trying to acquire an operation object; in this case as well, p helps the blocking operation.

In the example of Fig. 4, when op_2 discovers that it is blocked by op_1 with a higher counter, op_2 helps op_1 instead of waiting for it to complete. If op_1 is blocked by an operation that owns the operation object of op_3, while op_1 tries to reset op_3, then op_2 helps the blocking operation to proceed until it releases the operation object of op_3.

Due to helping, an operation may be executed by several *executing processes*, simultaneously. The process that invoked the operation is called its *initiator*.

5.1. Data structures

Since there are several processes executing an operation of p, some changes are required in the data structures of BLocalRMW and the way they are handled, to ensure that only one executing process performs each step of the operation.

The modified data structures appear in Fig. 7. The most important change is that *modifyDone* flag and the *conflict* attribute are visible to all processes executing the operation, i.e., kept in the shared memory. To avoid the *ABA problem*[6] of CAS and DCAS, we associate a monotonically increasing *ABA-prevention tag* with attributes that may hold the same value during an execution. This is the case for the owner attribute, which is reset when the object is released, and for conflict information, which is reset when the conflict is solved.

5.2. Implementation

LocalRMW combines the high-level scheme of Pseudocode 1 with the methods of Pseudocode 3 and Pseudocode 4. The main difference from BLocalRMW is that when an operation op is blocked by another operation op' with higher counter, op helps op' to proceed and release its data items (line 88). Additionally, if an executing process p of op tries to acquire the operation object of op' in order to reset op' and discovers it is owned by an operation op'', then p helps op'' (line 122).

As in BLocalRMW, the applyChangesAndRelease method applies the changes of the operation (line 50), sets *modifyDone* to TRUE (line 51), and releases the data items (line 52).

The acquireNextItem method checks if the current item is released (line 56) and tries to acquire it (line 58). Before doing so, it invalidates the conflict information, by "touching" the ABA value (line 57), so it is not set by another executing process that encountered contention on the same item. Finally, if the item is owned by the operation, the method returns success (line 60).

The handleConflict method handles the case where op is blocked by another operation op', indicated by the *conflict* attribute (line 69). The method reads the conflict information and the context of op' (line 70 and line 71). Then the method verifies that neither op nor op' changed round (line 72), as in BLocalRMW. If one of the round numbers changed, the method invalidates the context (line 73), so the operation object is not released by an executing process of op that is not aware that the conflict is resolved. Then the method resets the conflict information (line 74). If both round numbers did not change, the method compares the counter of op with the counter of op' and handles the conflict as in BLocalRMW; unless the counter of op' is higher than the the counter of op, in which case, op helps op' (line 88).

The reset method seizes the item held by the blocking operation (line 92) after reading it (line 91). If the item is owned by the operation in round number as specified in the input parameter (line 94), then the method increases the counter of the operation (line 95) and releases the data set of the blocking operation (line 96).

6 In the ABA problem [18], a process p may read a value A from some memory location l, then other processes change l to B and then back to A, later p applies CAS on l and the comparison succeeds whereas it should have failed.

Pseudocode 3 Methods for LocalRMW (continued in Pseudocode 4)

```
49: applyChangesAndRelease(Context ctx) {
50:     modify()
51:     WRITE(modifyDone, TRUE)
52:     releaseDataset(ctx.round)
53: }

54: boolean acqItem(Item item, Context ctx, Conflict cnfl) {
55:     owner ← READ(item.owner)
56:     if owner = ⊥ then                                                    // the item has no owner
57:         if CAS(conflict, ⟨⊥, ⊥, ⊥,cnfl.aba⟩, ⟨⊥, ⊥, ⊥,cnfl.aba+1⟩) then   // invalidate conflict
58:             item.acquire(owner, self, ctx.round, ctx.counter)            // acquire the item
59:     owner ← READ(item.owner)
60:     return (owner ≠ ⊥ and owner.op = self and owner.round = ctx.round)
61: }

62: increaseCounter(Context ctx) {
63:     CAS(context, ctx, ⟨ctx.counter+1,ctx.owner,ctx.round⟩)
64: }

65: initializeConflictInfo(Context ctx, Conflict cnfl) {
66:     CAS(conflict, ⟨⊥, ⊥, ⊥,cnfl.aba⟩, ⟨ctx.counter,ctx.round,owner,cnfl.aba+1⟩)
67: }

68: handleConflict(Context ctx, Conflict cnfl) {
69:     blkOp ← cnfl.blocking.op
70:     blkCnfl ← READ(blkOp.conflict)
71:     blkCtx ← READ(blkOp.context)
72:     if ⟨ctx.round,blkCtx.round⟩ ≠ ⟨cnfl.round,cnfl.blocking.round⟩ then
73:         CAS(op.context, ctx,⟨ctx.counter,⟨ctx.owner.op,ctx.owner.round,ctx.owner.aba+1⟩,ctx.round⟩)
74:         CAS(conflict, cnfl, ⟨⊥, ⊥, ⊥,cnfl.aba+1⟩)                        // reset conflict
75:         return
        // conflict with an operation with a lower counter
76:     if cnfl.counter > blkCtx.counter then
77:         if acqOperation(blkOp, blkCtx,ctx.round) then
78:             reset(cnfl.counter, cnfl.blocking, ⟨self,ctx.round,ctx.owner.aba⟩)
79:         return
80:     if ⟨ctx.owner.op,ctx.owner.round⟩ = ⟨self,ctx.round⟩ then
81:         if ⟨blkCtx.owner.op,blkCtx.owner.round⟩ ≠ ⟨self,ctx.round⟩ then
82:             CAS(context, ctx, ⟨ctx.counter,⟨⊥, ⊥,ctx.owner.aba+1⟩,ctx.round⟩)
83:             return
        // conflict with an operation with an equal counter
84:     if cnfl.counter = blkCtx.counter then
85:         if acqTwoOperations(self, ctx, blkOp, blkCtx, ctx.round) then
86:             reset(cnfl.counter, cnfl.blocking, ⟨self,ctx.round,ctx.owner.aba⟩)
        // conflict with an operation with a higher counter
87:     if cnfl.counter < blkCtx.counter then
88:         blkOp.executeIteraion(blkCnfl, blkCtx)
89: }

90: reset(int ctr, Owner blocking, Owner owner) {
91:     item ← READ(dataset[ctr])
92:     item.acquire(blocking, owner.op, owner.round, ctr)                   // seize the item
93:     itmOwner ← READ(item.owner)
94:     if ⟨itmOwner.op, itmOwner.round⟩ = ⟨owner.op, owner.round⟩ then
95:         CAS(context, ⟨ctr,owner,owner.round⟩, ⟨ctr+1,owner,owner.round⟩)
96:     blocking.op.releaseDataset(blocking.round)
97: }

98: releaseDataset(int rnd) {
99:     ctx ← READ(context)
100:    for i = 0 to dataset.size−1 do
101:        item ← READ(dataset[i])
102:        itmOwner ← READ(item.owner)
103:        CAS(item.owner, ⟨self,rnd,itmOwner.aba⟩, ⟨⊥, ⊥, itmOwner.aba+1⟩)
        // increase round number, reset counter, release operation object
104:    CAS(context, ⟨ctx.counter,ctx.owner,rnd⟩, ⟨0,⟨⊥, ⊥,ctx.owner.aba+1⟩,rnd+1⟩)
105: }
```

Pseudocode 4 LocalRMW (continued from Pseudocode 3)

```
106: boolean acqOperation(Operation op, Context ctx, int rnd) {
107:     CAS(op.context, ⟨ctx.counter, ⟨⊥, ⊥,ctx.owner.aba⟩,ctx.round⟩, ⟨ctx.counter,⟨self,rnd,ctx.owner.aba+1⟩,ctx.round⟩)
108:     return op.verifyOwner(self, rnd, ctx.round)
109: }

110: boolean acqTwoOperations(Operation op1, op2, Context cx1, cx2, int rnd) {
111:     DCAS(op1.context,                                   op2.context,
             ⟨cx1.counter,⟨⊥, ⊥,cx1.owner.aba⟩,cx1.round⟩,   ⟨cx2.counter,⟨⊥, ⊥,cx2.owner.aba⟩,cx2.round⟩,
             ⟨cx1.counter,⟨self,rnd,cx1.owner.aba+1⟩,cx1.round⟩,  ⟨cx2.counter,⟨self,rnd,cx2.owner.aba+1⟩,cx2.round⟩)
112:     return op1.verifyOwner(self, rnd, cx1.round) and op2.verifyOwner(self, rnd, cx2.round)
113: }

114: boolean verifyOwner(Operation op, int opRnd, int rnd) {
115:     ctx ← READ(context)
116:     ownerOp ← ctx.owner.op
117:     if ⟨ownerOp,ctx.owner.round⟩ = ⟨op,opRnd⟩ and ctx.round = rnd then
118:         return TRUE                                                  // indicate success
119:     if ownerOp ≠ ⊥ and ctx.round = rnd then                          // other operation owns this operation
120:         ownerCnfl ← READ(ownerOp.conflict)
121:         ownerCtx ← READ(ownerOp.context)
122:         ownerOp.executeIteraion(ownerCnfl,ownerCtx)                  // help execute the operation
123:     return FALSE                                                     // indicate failure
124: }

     // Item's method
125: acquire(Owner ownr, Operation op, int rnd, int ctr) {
126:     ctx ← READ(op.context)
127:     if ⟨ctx.counter,ctx.owner.op,ctx.owner.round,ctx.round⟩ = ⟨ctr,op,rnd,rnd⟩ then
128:         CAS(owner, ownr, ⟨op,rnd,ownr.aba+1⟩)                        // acquire the item
129: }
```

As in BLocalRMW, the releaseDataset method reads (line 102) and releases (line 103) every item that is owned by the operation, and then resets the context of the operation (line 104). However, in order to avoid modifications when the operation changes its round, the round number given as input parameter is used when releasing the items and updating the context of the operation.

The acqOperation method applies CAS to acquire the operation object given as input parameter (line 107), and verifies that this operation owns the given operation object (line 108). The acqTwoOperations method is similar, except for using DCAS to acquire the *two* operation objects (line 111), and verifying that this operation owns *both* operations object (line 112).

5.3. Safety

We prove that LocalRMW is linearizable by showing that the executing processes are synchronized. Appendix B shows the following invariants of LocalRMW:

Invariant 1: When *op* increases its counter it holds its operation object (Lemma 13).

Invariant 2: Before an operation *op* acquires all its data items, *op* releases an item only when another operation holds the operation object of *op* while resetting *op* (Lemma 14(1)).

Invariant 3: While *op* holds its operation object and its counter is *i* in the *r*-th round, the first *i* items of *op* are owned by *op* (Lemma 14(2)).

We identify, for each operation, a *linearization point* in its interval, so that the operation appears to occur atomically at this point. The linearization point of an operation is when it sets the *modifyDone* flag to TRUE (line 51).[7] A concrete implementation of the *k*RMW operation defines the modify method, and must ensure that the data items are changed only if the *modifyDone* flag of *op* is not TRUE, i.e., before the linearization point. Appendix A presents an example of modify method for the common and important case of *k*CAS.

The value of a data item is changed by *op* only when *op* owns all its data items. Thus, operations apply their changes in isolation, and can be considered to take effect atomically at the linearization point. We sketch the proof and defer the complete proof to Appendix B.

Theorem 8 (*Linearizability*)**.** *LocalRMW is linearizable.*

Sketch of proof. Invariant 1 implies that when an operation *op* increases its counter to *k* it holds its operation object, and Invariant 3 implies it holds all its data items.

[7] This is similar to how linearization points are defined in [3].

An executing process p' of op reads the context of op before or after the counter is set to k. If the value is lower than k, when p' attempts to change the context after the counter increases and before the *modifyDone* flag of op is, it fails. Invariant 2 implies that p' does not release any item of op. If the value is k, then p' invokes the modify method and sets the *modifyDone* flag to TRUE. Only then p' releases the items of op in the releaseDataset method, and changes the context of op.

Since modify is invoked after op acquired all its data items, the implementation ensures that the data items are changed only if the *modifyDone* flag of op is not TRUE, and that op holds all its data items when applying the changes. □

5.4. Progress and locality

The locality and progress proofs of LocalRMW are similar to those of BLocalRMW. Lemmas 1–6 can be adapted to hold also for LocalRMW. Then, we prove that if a process p takes many steps in executing op, it will eventually get to help any process that might be blocking it. In this way, helping "simulates" a scenario in which the initiators of nearby operations do not stop taking steps, alleviating the effects of blocking. Specifically, the next lemma proves that after a finite number of steps by p, there is progress in the $2k$-neighborhood of op, and the following lemma uses it to show that eventually, some operation in the $3k$-neighborhood of op completes.

The proof relies on a parameterized notion of an increasing–decreasing chain: in an $\langle l, m\rangle$-*increasing–decreasing* chain, the length of the increasing sub-chain is l and the counter of the first operation in the decreasing chain op is m.

Lemma 9. *Given an operation op in a configuration C, consider an $\langle l, m\rangle$-increasing–decreasing chain $\mathcal{IDC}$ from op. There is an execution interval α from C that includes $O(l + m + k)$ steps of an executing process of op, such that either α is a big step of the $\mathcal{IDC}$ or α contains a progress step of operation in the $2k$-neighborhood of op.*

Proof. We prove the lemma by double induction on l and m.

First consider the case that $l = 1$ and $m \geq 0$. The base case is a $\langle 1, 0\rangle$-increasing–decreasing chain, which includes only op, and the lemma clearly holds in any execution with one step of an executing process of op.

Next assume the claim holds for a $\langle 1, m'\rangle$-increasing–decreasing chain with any m', $0 \leq m' < m$. Consider an operation op with $c(op) \geq m$ that is m-delayed by another operation op' that is the first operation in the decreasing chain, with $c(op') = m$. When op discovers that it is blocked by op' with lower or equal counter (line 76 or line 84), op tries to acquire the operation object of op' (line 77 or line 85). Then op verifies that it owns the operation object of op' (line 108 or line 112). If it succeeds to acquire the operation object of op' (line 117), op resets op' (line 78 or line 86); op increases its counter and completes the reset within $O(k)$ steps, and the claim is satisfied.

Otherwise, op fails to acquire both the operation object of op and op'. Assume op discovers that some operation op'' acquired one of the operation objects (line 119). If op'' is resetting op or op', op helps op'' to complete the reset and increase its counter within $O(k)$ steps, and the claim holds since op'' is in the 2-neighborhood of op.

If op discovers that op' holds the operation object of op' then op helps op'. We note that if no operation in the $2k$-neighborhood has a progress step, op' has a $\langle 1, m'\rangle$-increasing–decreasing chain, where $m' < m$. By the inductive assumption, after $O(l + m' + k)$, which is $O(l + m + k)$ steps of the executing process one of the properties holds.

If the round number of op increases, then op was reset increasing the counter of the resetting operation and the claim is satisfied. If the counter of op' increases the claim is also satisfied. Otherwise, op fails to acquire the operation objects of op and op', yet no operation holds them when the executing process checks the owners (line 115). If this occurs more then once, while none of the above scenarios apply, then it implies that op' releases and re-acquires its operation object. Hence, either some operation op'' blocking op' completed or was reset (increasing the counter of the resetting operation) and is no longer blocking op', or op' was reset (increasing the counter of the resetting operation) and is no longer blocked by op''. Hence some operation in the 3-neighborhood of op has a progress step, satisfying the claim.

To prove the claim for any $l > 1$, assume it holds for any $\langle l - 1, m\rangle$-increasing–decreasing chain with $m \geq 0$. Consider an operation op that is blocked by an operation op' with $c(op') > c(op)$, which is blocked by an $\langle l - 1, m\rangle$-increasing–decreasing chain. When op discovers that op is blocked by op' with higher counter (line 87), it helps op' (line 88) and becomes an executing process of op'. By the inductive assumption, after $O(l - 1 + m + k)$ steps of the executing process one of the properties holds. □

Recall that n_d is the number of operations in the d-neighborhood of op.

Lemma 10. *After $n_{2k}n_{k-1}k^2$ steps of an executing process of operation op, some operation in the $3k$-neighborhood of op completes.*

Proof. Consider an operation op in a configuration C, with an $\langle l, m\rangle$-increasing–decreasing chain from op, in C. Note that op by itself is a $\langle 1, m\rangle$-increasing–decreasing chain, where m is the counter of op. Lemma 9 implies that there is an execution interval α_1 starting at C, in which either an executing process of op takes $O(k)$ steps and some operation in the $2k$-neighborhood of op has a progress step in α_1, or α_1 contains $O(1)$ big steps of the increasing–decreasing chain.

Consider such $n_{2k}n_{k-1}k^2$ consecutive execution intervals $\alpha_1, \alpha_2, \cdots$. Lemmas 6 and 9 (adapted for LocalRMW), imply $n_{2k}n_{k-1}k^2$ progress steps in the $2k$-neighborhood of op. Since the number of operations in the $2k$-neighborhood of op is at most n_{2k}, some operation in this neighborhood has $n_{k-1}k^2$ progress steps and by Lemma 4 (adapted for LocalRMW), some operation in the $3k$-neighborhood of op completes. □

Afek et al. [1] define the *local step complexity* to capture the locality of wait-free implementations. Roughly, an implementation has *d-local step complexity* if the step complexity of an operation is bounded by a function of the number of operations in its d-neighborhood. A dynamic variant of this definition can be considered as the quantitative analogue of the local nonblocking definition, and indeed Lemma 10 implies that LocalRMW has $3k$-local step complexity. Moreover, since the number of processes is finite, it follows that an operation has an infinite number of steps without completing only if infinitely many operations complete in its $3k$-*neighborhood*.

Corollary 11 (*Local Nonblocking*)**.** *LocalRMW is* $3k$*-local nonblocking.*

Afek et al. [1] also define the notion of *local contention*, which captures the locality of memory contention created by an implementation. Roughly, an implementation has *d-local contention* if two operations accessing the same memory location simultaneously are at distance $\leq d$.

Lemma 12. *LocalRMW has* $(4k - 6)$*-local contention.*

Proof. We show that the initiator of an operation op_1 accesses the same memory location as the initiator of an operation op_2, only if op_2 is in the $(4k - 6)$-neighborhood of op_1.

An initiator accesses an item t or an operation object o only while helping an operation op_1 such that o is the operation object of op_1 or t is in the data set of op_1. It can be proved by induction that an operation op_1 only helps another operation op_2 if op_2 is on an increasing–decreasing chain from op_1, or if op_2 holds an operation on an increasing–decreasing chain from op_1.

Thus, an operation op_1 contends with another operation op_2 only if they are helping an operation op_3 (which may be either op_1 or op_2). The operation op_3 is on an increasing–decreasing chain from both op_1 and op_2, or holds an operation on such a chain, thus, it is within the $(2k - 3)$-neighborhood of both op_1 and op_2, and the distance between op_1 and op_2 is at most $4k - 6$. □

6. Discussion

We have presented a highly concurrent implementation of kRMW, with improved throughput even when there is contention; it stores a constant amount of information per data item, independently of k. The implementation can be made wait-free, without sacrificing its locality properties, by applying a known technique [2] (see [1]).

We believe that DCAS provides critical leverage allowing to implement kRMW, for any $k > 2$, with locality that is difficult, perhaps impossible, to obtain using only CAS. Currently, few architectures provide DCAS in hardware, but DCAS is an ideal candidate to be supported by *hardware transactional memory* [15,20], being a short transaction with static data set of minimal size (two). Alternatively, DCAS can be simulated in software from CAS [3,9], or by applying a simple randomized algorithm [11].

It is interesting to obtain locality properties that are independent of n, without using DCAS. A lower bound presented in [3] indicates that this might be impossible, but the exact bounds and tradeoffs should be explored. Even more intriguing is to investigate whether $O(k)$ is the best locality that can be achieved, even with DCAS.

A flexible dynamic variant of our implementation can serve as the basis for a *dynamic* STM acquiring ownership at *encounter-time*. Encounter-time semantics enables early detection of conflicts, particularly important when transactions are long; on the other hand, early conflict detection may lead to unnecessary aborts [8,10]. It is also possible to acquire ownership at *commit-time* with a *speculative execution* scheme [8], first accumulating the data items the operation needs to access, and then acquiring them, with the algorithm presented here. Realizing a full-fledged STM requires to address many additional issues, e.g., memory management, handling read-only data, and optimizing the common case, which are left for future research.

Acknowledgements

We thank Vincent Gramoli, Rachid Guerraoui, David Hay, Danny Hendler, Alex Kogan, Alessia Milani, Alex Shraer and the anonymous referees for helpful comments.

Appendix A. *k*CAS Implementation

The kCAS operation extends and refines the implementation of the kRMW operation. In addition to the attributes of an *Operation* instance, each kCAS operation instance contains two additional arrays of size k: *expValues* and *newValues*. The *result* flag indicates whether the operation succeeded to update the new values of the items, like in the unary CAS primitive. When invoking a kCAS operation, the process executes the rmw method, during which the *result* flag is set, and completes the operation by returning the value written in *result*.

Pseudocode 5 presents the modify method for a kCAS operation. A process executing the modify method first verifies that all the values of the items in *op*'s data set are consistent with their expected values (lines 133–137), then the *result* flag is set by the result of the consistency test (line 138). If the consistency test is successful (line 139) and the modifications of the operation have not been applied yet (line 144), then the values of the items in *op*'s data set are set to their new values, respectively (lines 140-146). ABA-prevention tags are associated with the data the operation is modifying, to ensure that

Pseudocode 5 The modify method for *k*CAS.

```
  class kCAS(extends Operation) {
      Data expValues[k]
      Data newValues[k]
      Result result                                   // initialized to NULL, set to TRUE or FALSE
  }
130: kCAS::modify() {
131:     Data oldValues[k]
132:     res ← TRUE
133:     for i = 0 to k − 1 do
134:         item ← READ(dataset[i])
135:         oldValues[i] ← READ(item.data)
136:         exp ← READ(expValues[i])
137:         res ← res and (oldValues[i].val = exp)
138:     CAS(result, NULL, res)
139:     if READ(result) = FALSE then return
140:     for i = 0 to k − 1 do
141:         item ← READ(dataset[i])
142:         exp ← READ(expValues[i])
143:         new ← READ(newValues[i])
144:         if READ(modifyDone) = TRUE then return
145:         if exp ≠ new then
               // write modified values
146:           CAS(item.data,⟨exp,oldValues[i].aba⟩,⟨new,oldValues[i].aba+1⟩)
147: }
```

each operation modifies its data set only once while owning all items in the data set, and before the *modifyDone* flag of the operation is set to TRUE (for a detailed proof see [17]).

Appendix B. Safety proof

We first provide definitions required for the proofs. An operation *op* is in a $\langle c, l, r\rangle$-*context* if the *context* of *op* is $\langle c, l, r\rangle$. An operation is in the *r-th round* if it is in a $\langle c, l, r\rangle$-context for some *c* and *l*. An item object *t* or an operation object *o* are $\langle op, r\rangle$-*owned* if the *owner* of *t* or the *context.owner* of *o* respectively, are equal to $\langle op, r, aba\rangle$, for some *aba*. A process *p* $\langle op, r\rangle$-*acquires* an item *t* (operation object *o*) if *p* applies a CAS to *t* (*o*) such that prior to applying the CAS *t* (*o*) is not $\langle op, r\rangle$-owned, and after which *t* (*o*) is $\langle op, r\rangle$-owned. Similarly, we define the process that $\langle op, r\rangle$-*releases* an item (operation object).

We prove that an operation applies its changes exactly once after acquiring all its data items; this claim is used in the proof of Lemma 14.

Lemma 13. *For every i,* $0 \leq i < k$*, l and r, when op shifts from the* $\langle i, l, r\rangle$*-context to the* $\langle i + 1, l, r\rangle$*-context, op is* $\langle op, r\rangle$*-owned.*

Proof. An executing process *p* increases the counter of *op* either in line 63, or in line 95 after resetting another operation. In the first case, *p* increases the counter (line 63) only if *op* is $\langle op, r\rangle$-owned. In the second case, one of the input parameters for the reset method is a pair, which contains the operation object of *op* and its round number *r*, as *p* previously read them; then, *p* shifts *op* to the $\langle i + 1, l, r\rangle$-context, only if it is $\langle op, r\rangle$-owned.

In both cases, the CAS is successful only if *op* is $\langle op, r\rangle$-owned and the ABA value read is not changed, implying the lemma. □

Lemma 14. *The following properties hold for every* $r \geq 0$*, i,* $0 \leq i < k$*, and every process p:*

1. *If op is in a* $\langle i, l, r\rangle$*-context and p* $\langle op, r\rangle$*-releases an item then op is* $\langle op', r'\rangle$*-owned and* $op' \neq op$*.*
2. *If op is in a* $\langle i,\langle op, r, a\rangle,r\rangle$*-context, then the j-th item of op is* $\langle op, r\rangle$*-owned, for every j,* $0 \leq j < i$*.*
3. *If p* $\langle op, r\rangle$*-acquires an item then op is in the r-th round and is* $\langle op, r\rangle$*-owned.*

Proof. The proof is by induction on the execution order. The base case is the empty execution where all the properties are vacuously satisfied. Next we prove the induction step for each property; we only review steps that are relevant to the properties. We say that an owner $\langle cop, cr, a\rangle$ is *blocking op in its r-th round on the i-th item of op* if the *conflict* of *op* is equal to $\langle i, r, \langle cop, cr, a\rangle, aba\rangle$, for some *aba* value.

Property 1: An executing process $\langle op, r\rangle$-acquires an item only if it is in the data set of *op*. By Property 3, an executing process $\langle op, r\rangle$-acquires the item only if *op* is in the *r*-th round. Since $i < k$, *p* does not apply the releaseDataset method in the execute method of *op*. Thus, *p* $\langle op, r\rangle$-releases an item *t* in the reset method of another operation *op*′. Prior to $\langle op, r\rangle$-releasing *t* (lines 78 or 86), *p* acquires *op* (lines 77 or 85) and verifies that *op* is in the *r*-th round and is $\langle op', r'\rangle$-owned (line 117).

If another executing process, p', $\langle op', r'\rangle$-releases op then it applied line 104, and op is no longer in the r-th round. Prior to this, an item, t, in op's data set is either not $\langle op, r\rangle$-owned or p' $\langle op, r\rangle$-releases t (line 103). Whether op is $\langle op', r'\rangle$-owned or is not in the r-th round, Property 3 implies that no executing process $\langle op, r\rangle$-acquires any item. Thus, p fails to $\langle op, r\rangle$-release t (in line 92 or 96).

Property 2: An executing process p increases the counter of op and shifts to the $\langle i, \langle op, r, a\rangle, r\rangle$-context either in line 63 or in line 95 after resetting another operation. Since the CAS is successful, op applies the CAS in a $\langle i-1, \langle op, r, a\rangle, r\rangle$-context. By the inductive assumption for every j, $0 \leq j < i-1$, the j-th item of op is $\langle op, r\rangle$-owned. Next we prove that the $(i-1)$-th item is also $\langle op, r\rangle$-owned.

If p increases the counter in line 63, then it reads the $(i-1)$-th item of op and the item's owner (line 55) and verifies that the $(i-1)$-th item is $\langle op, r\rangle$-owned (line 60). If p increases the counter in line 95, it reads the $(i-1)$-th item of op (line 91) and the item's owner (line 93) and verifies that the $(i-1)$-th item is $\langle op, r\rangle$-owned (line 94). In both cases, since the ABA value in the context of op has not changed after p reads it, no operation op' other than op $\langle op', r'\rangle$-owned op. By Property 1, no executing process $\langle op, r\rangle$-releases the $(i-1)$-th item.

Hence, when op increases the counter and shifts to a $\langle i, \langle op, r, a\rangle, r\rangle$-context, the j-th item of op is $\langle op, r\rangle$-owned for every j, $0 \leq j < i$. Since $i < k$, Property 1 implies that while op is in the $\langle i, \langle op, r, a\rangle, r\rangle$-context and is $\langle op, r\rangle$-owned, no executing process $\langle op, r\rangle$-releases any item.

It is left to handle the scenario in which an executing process $\langle op, r\rangle$-releases op and another (consider the first) executing process p' $\langle op, r'\rangle$-acquires op afterwards. Assume p' shifts op to a $\langle i', \langle op, r', a'\rangle, r'\rangle$-context. If while op is not $\langle op, r\rangle$-owned some executing process $\langle op, r\rangle$-releases an item then, Property 1 implies that op was $\langle op'', r''\rangle$-owned ($op'' \neq op$), and some executing process $\langle op'', r''\rangle$-releases op (in line 104), before p' $\langle op, r'\rangle$-acquires op. Thus, $i' = 0$, and the claim vacuously holds. Otherwise, by Property 1, while op is not $\langle op, r\rangle$-owned, op was not reset, and no executing process $\langle op, r\rangle$-releases any item. Hence, $r' = r$, $i' = i$, and the property holds.

Property 3: Assume, towards a contradiction, that the owner or round number of an operation nop change after an executing process p verifies that nop is in a $\langle ctr, \langle nop, nr, la\rangle, nr\rangle$-context (line 127) and before p $\langle nop, nr\rangle$-acquires an item t in the acquire method (line 128).

The first case is when an executing process, p', $\langle nop, nr\rangle$-releases nop for the first time in line 104. If p' applies line 104 while executing the reset method of another operation op'. Before the reset (line 78 or 86), p' verified that nop is $\langle op', r'\rangle$-owned (line 77 or 85). Thus, this is not the first time an executing process $\langle nop, nr\rangle$-releases nop after p read the context of nop, a contradiction. Otherwise, p' applied line 104 in the execute method of nop after reading $c(nop) = k$ while p reads $c(nop) = ctr < k$. By Property 2, when the counter is set to $ctr + 1$, t is $\langle nop, nr\rangle$-owned, and the CAS applied by p so as to acquire t fails since at least the ABA value changes, a contradiction.

The second case is when an executing process p' changes the context by applying line 82. If p invokes the acquire method in line 58, then p reads that no owner is blocking nop while p' reads that some $\langle cop, cr, ca\rangle$ owner is blocking nop. Consider the executing process p'' that set *conflict* of nop to this value. Since p acquires t, no other operation acquires t after p reads it and before p $\langle nop, nr\rangle$-acquires t. Hence, the interleaving of the steps is as follows: p'' reads the *conflict* of nop, including its ABA value. Then, p'' reads the owner on t (line 55) and discovers that t is $\langle cop, cr\rangle$-owned. Afterwards, p reads that no operation owns t (line 56) and invalidates *conflict* of nop (by "touching" the ABA value). Hence, p'' cannot set *conflict* (line 66), since at least the ABA value changes when p invalidates *conflict*, a contradiction.

Otherwise, p invokes the acquire method in line 92. In this case both p and p' read that nop is blocked. p reads that the owner $\langle cop, cr, a\rangle$ is blocking nop on its i-th item in the nr-th round and acquires the operation object of cop in configuration C so as to reset cop. Assume p' reads the same value in *conflict* of nop. That is, p' reads that the owner $\langle cop, cr, a\rangle$ is blocking nop on its i-th item in the nr-th round. Claim 15 implies that p' cannot apply line 82.

Claim 15. *During the execution interval α that starts in C and ends when the reset is completed, p' does not $\langle op, r\rangle$-release op in line 82.*

Proof. If p $\langle op, r\rangle$-acquires cop and op atomically in line 85 then either p' discovers that both cop and op are $\langle op, r\rangle$-owned or it fails to $\langle op, r\rangle$-release op since at least the ABA value has changed.

Otherwise, p $\langle op, r\rangle$-acquires cop in line 77. p reads the context of cop (line 71) and discovers that $i > c(cop)$ (line 76). By Lemma 13, if the counter of cop increases then cop is $\langle cop, cr'\rangle$-owned for some cr', after cop was $\langle op, r\rangle$-released in line 104 at the end of α. So, during α, $i > c(cop)$ and p' does not get to apply line 82 successfully. □

Assume p' reads a different value in *conflict* of nop. That is, p' reads that some lock $\langle cop', cr', a'\rangle$ is blocking nop on its i'-th item in the nr'-th round. If $\langle cop', cr', a'\rangle$ was blocking nop prior to $\langle cop, cr, a\rangle$, then some executing process invalidates the context of nop (line 73) before resetting *conflict* of nop (line 74) and setting it to $\langle cop, cr, a\rangle$. Hence, p' cannot change the context of nop, a contradiction. Otherwise, $\langle cop, cr, a\rangle$ was blocking nop prior to $\langle cop', cr', a'\rangle$.

Claim 16. *During the execution interval α that starts in C and ends when the i-th item of op is $\langle op, r\rangle$-owned, the round numbers of op and cop do not change.*

Proof. Consider the first event e changing the round number of cop by an executing process p'' in line 104. If p'' increases the round number in the execute method of cop then the counter of cop increased to the size of cop's data set. By Lemma 13, when the counter increases, cop is $\langle cop, cr'\rangle$-owned for some cr'. Otherwise, p' increases the round number of cop in the reset

method of another operation op', after verifying that cop is $\langle op', r'\rangle$-owned. Both cases imply that cop was $\langle op, r\rangle$-released first in line 104, which implies an increase in the round number of cop, contradicting the assumption that e is the first event changing the round number.

Consider the first event e' changing the round number of op applied by an executing process p'' in line 104. If p'' increases the round number of op after executing the modify method then the counter of op increased to k. By Property 2, when the counter is set to $i+1$ the i-th item is already $\langle op, r\rangle$-owned, and the claim holds. Otherwise, p' increases the round number while executing the reset method of another operation op' after verifying that op is $\langle op', r'\rangle$-owned. If some executing process $\langle op, r\rangle$-releases op in line 104 prior to e', then this implies an increase in the round number of op, contradicting the assumption that e' is the first event changing the round number. Otherwise, some executing process $\langle op, r\rangle$-releases op in line 82 during α, contradicting Claim 15. □

By Claim 16, neither nop nor cop change their round numbers. Hence, no executing process can set *conflict* of nop (line 74) to $\langle cop', cr', a'\rangle$ before p $\langle nop, nr\rangle$-acquires t. □

Theorem 8 *LocalRMW is linearizable.*

Proof. Lemma 13 implies that when an operation op shifts to the $\langle k, l, r\rangle$-context, it is $\langle op, r\rangle$-owned, i.e., $l = \langle$op,r,a$\rangle$, and Lemma 14(2) implies that op owns all its data items.

An execution process p' of op reads the $\langle ic, il, ir\rangle$-context of op either before or after the shift, i.e., either $ic < k$ or $ic = k$.

If $ic < k$, any attempt to change the context by p' after the shift and before the *modifyDone* flag of op is set to TRUE fails: p' fails to apply the CAS in lines 63, 73, 82 using ic since the counter has changed. ic is also used in line 95 while executing the reset method (lines 78, 86). Changing the context in lines 107 and 111, while trying to acquire the operation, (lines 77, 85), fails since the operation is $\langle op, r\rangle$-owned and the owner is not equal to $\perp$. Lemma 14(1) implies that p' does not $\langle op, r\rangle$-release any item of op.

If $ic = k$, then p' invokes the modify method (line 50) and sets the *modifyDone* flag to TRUE (line 51). Only then p' $\langle op, r\rangle$-releases the items of op (line 103 in the releaseDataset method, called in line 52), and changes the context of op (line 104 in the releaseDataset method).

Since modify was invoked after op acquired all its data items,the implementation ensures that the data items are changed only if the *modifyDone* flag of op is not TRUE, hence, guaranteeing that while applying the changes all the items in the data set of op are $\langle op, r\rangle$-owned. □

References

[1] Y. Afek, M. Merritt, G. Taubenfeld, D. Touitou, Disentangling multi-object operations, in: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing, PODC, 1997, pp. 111–120.
[2] J.H. Anderson, M. Moir, Universal constructions for multi-object operations, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, 1995, pp. 184–193.
[3] H. Attiya, E. Dagan, Improved implementations of binary universal operations, Journal of the ACM 48 (5) (2001) 1013–1037.
[4] H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, second edition, John Wiley & Sons, 2004.
[5] G. Barnes, A method for implementing lock-free shared-data structures, in: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA, 1993, pp. 261–270.
[6] M. Choy, A.K. Singh, Efficient fault-tolerant algorithms for distributed resource allocation, ACM Transactions on Programming Languages and Systems 17 (3) (1995) 535–559.
[7] R. Cole, O. Zajicek, The APRAM: incorporating asynchrony into the PRAM model, in: Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA, 1989, pp. 169–178.
[8] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Proceedings of the 20th International Symposium on Distributed Computing, DISC, 2006, pp. 194–208.
[9] F.E. Fich, V. Luchangco, M. Moir, N. Shavit, Obstruction-free step complexity: Lock-free dcas as an example, in: Proceedings of the 19nd International Symposium on Distributed Computing, DISC, 2005, pp. 493–494.
[10] V. Gramoli, D. Harmanci, P. Felber, On the input acceptance of transactional memory, Parallel Processing Letters 20 (1) (2010) 31–50.
[11] P.H. Ha, P. Tsigas, M. Wattenhofer, R. Wattenhofer, Efficient multi-word locking using randomization, in: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, PODC, 2005, pp. 249–257.
[12] T. Harris, K. Fraser, Language support for lightweight transactions, ACM SIGPLAN Notices 38 (11) (2003) 388–402.
[13] T.L. Harris, K. Fraser, I.A. Pratt, A practical multi-word compare-and-swap operation, in: Proceedings of the 16th International Conference on Distributed Computing, DISC, Springer-Verlag, London, UK, 2002, pp. 265–279.
[14] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing, PODC, 2003, pp. 92–101.
[15] M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, ACM SIGARCH Computer Architecture News 21 (2) (1993) 289–300.
[16] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.
[17] E. Hillel, Concurrent data structures: methodologies and inherent limitations, Ph.D. Thesis, Technion—Israel Institute of Technology, 2011.
[18] IBM. IBM System/370 Extended Architecture, Principle of Operation, 1983. IBM Publication No. SA22-7085.
[19] A. Israeli, L. Rappoport, Disjoint-access-parallel implementations of strong shared memory primitives, in: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, PODC, ACM, New York, NY, USA, 1994, pp. 151–160.
[20] R. Rajwar, J.R. Goodman, Transactional lock-free execution of lock-based programs, ACM SIGPLAN Notices 37 (10) (2002) 5–17.
[21] D.J. Rosenkrantz, R.E. Stearns, I. Philip, M. Lewis, System level concurrency control for distributed database systems, ACM Transactions on Database Systems 3 (2) (1978) 178–198.
[22] C.J. Rossbach, H.E. Ramadan, O.S. Hofmann, D.E. Porter, B. Aditya, E. Witchel, TxLinux and MetaTM: transactional memory and the operating system, Communications of the ACM 51 (9) (2008) 83–91.
[23] W.N. Scherer III, M.L. Scott, Advanced contention management for dynamic software transactional memory, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC, 2005, pp. 240–248.

[24] J. Schneider, R. Wattenhofer, Bounds on contention management algorithms, in: Proceedings of the 20th International Symposium on Algorithms and Computation, ISAAC, 2009, pp. 441–451.
[25] N. Shavit, D. Touitou, Software transactional memory, Distributed Computing 10 (2) (1997) 99–116 (special issue).
[26] J. Turek, D. Shasha, S. Prakash, Locking without blocking: making lock based concurrent data structure algorithms nonblocking, in: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS, 1992, pp. 212–222.